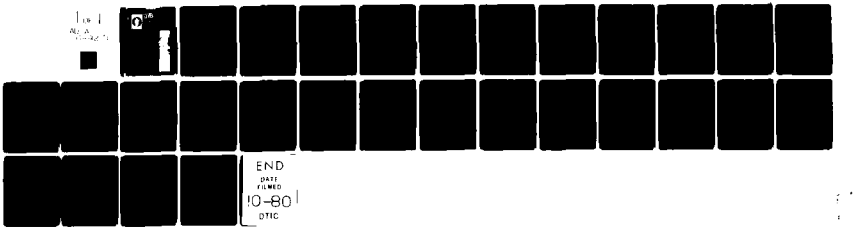
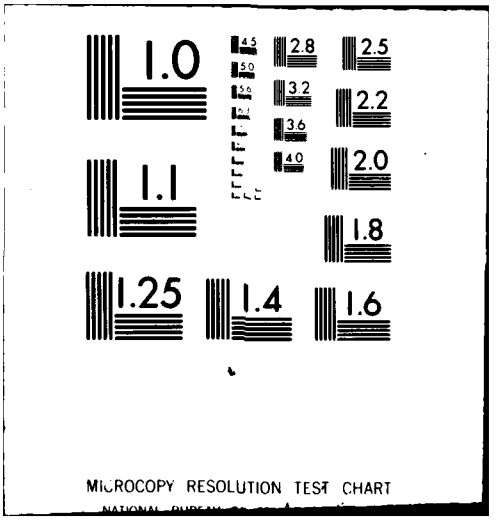
AD-A089 271 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
AN OVERVIEW OF RELIABLE COMPUTER SYSTEM DESIGN,(U)
MAY 80 J A MCDERMID
UNCLASSIFIED RSRE-80003 DRIC-BR-75035 NL



END
DATE FILMED
10-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

Procurement Executive, Ministry of Defence
Royal Signals and Radar Establishment, Christchurch

(11) May 80

(12) 30

Report No 80003

(6)

AN OVERVIEW OF RELIABLE COMPUTER SYSTEM DESIGN

by

(10) J. A. McDermid

(18) DRIC

(19) BR-75035

ABSTRACT

This paper was produced to support a series of lectures on reliable computer system design given at a NATO ASI summerschool on multiple processor computers. The paper was intended to be fairly self contained, but it does lack a description of a practical fault tolerant system. This omission was made because one of the other lectures (the one on the dual matched processor system referred to in section 7.5) analysed a typical system in some detail. Perhaps the best example of a practical fault tolerant system is the "Tandem 16" (See footnote) multi computer produced by Tandem Computers Limited. Any other sins of omission or comission are unintentional.

The paper presents an overview of reliable computer system design. It attempts to provide a pragmatic guide to redundancy and recovery, but does not give a very thorough discussion of either the theory or philosophy of reliable systems. The paper introduces and defines the basic concepts of reliability, and describes the basic mechanisms for achieving fault tolerance. It compares the attributes of multi processor and multi computer systems from the point of view of reliability. It describes in some detail techniques for achieving tolerance to both hardware and software faults. The paper concludes by outlining some of the major unsolved problems of reliable system design.

(14) RSRE-80003

The hardware is described in:
"A Fault - Tolerant Computing System", by J A Katzman, and the software in:
"A Non - Stop Operating System", by J F Bartlett.
Both these papers were presented at the Hawaii International Conference on System Sciences in 1978.

May 1980

1) OBJECTIVES AND FUNDAMENTAL CONCEPTS

1.1) Introduction

Computers are often embedded in systems on which great reliance is placed, such as in spacecraft flight control systems. Reliance may be regarded as the probability with which we would like the system to work continuously (the requirement), whereas the reliability is the probability of correct working, which we achieve in the implemented system (a function of the design). Our main objective in studying computer system reliability is to reach a state of knowledge, from which we can design systems which we can show meet the requirement for reliance.

Reliance is usually expressed as the minimum acceptable probability of the system giving continuous, correct service, for some particular mission time. A particular system is obviously suitable for an application if its inherent reliability - the probability that a failure does not occur - is higher than the reliance which we require. The study of reliable systems is interesting because there are many instances where the inherent reliability is lower than the reliance we require, so we have to devise measures for increasing the overall system reliability.

The availability - the probability that a system will provide a requested service - is also closely related to reliability, but the difference is significant. A system which fails frequently, but which can rapidly be restarted will have low reliability, but may well have high availability. We will be concerned here with systems with high reliability, not high availability, but many of the techniques used for improving reliability will also help in improving availability. It is generally easier to achieve high availability than high reliability.

1.2) Fault Tolerance and Fault Intolerance

If the inherent reliability of the system is less than the reliance we wish to place on it, then we can try to remedy this by two different means. We can:

> Improve the inherent reliability of the components to achieve the desired system reliability - this is known as Fault Intolerance, and/or we can
>
> Design the system so that it will still carry out its function despite the failure of some of its components thereby making the operational reliability greater than the inherent reliability - this is known as Fault Tolerance.

The use of these two techniques is not mutually exclusive, and in most practical systems both will be used. However our primary interest here is in techniques for achieving fault tolerance, and little time will be spent in discussing ways of improving the inherent reliability of components.

1.3) Classification of Failures

Failures are usually classified by the consequences of them occurring. The more serious the consequences of failure the greater is the reliance we have to place on the system, hence the greater is the reliability which we must achieve. The following three examples give some idea of the

range of requirements for reliance:

Student computer service:
Failure has a nuisance value only - a highly available, rather than highly reliable system is probably adequate.

A steel mill, or an Air Traffic Control (ATC) system:
Failure will cause loss of life and/or destruction of expensive equipment - high reliability, possibly with graceful degradation of system function is required.

Nuclear reactor:
A failure here may be catastrophic - the normal requirement is to stop safely and quickly as there will probably be insufficient time to use fault tolerant measures (to ensure that normal operation continues) before failure occurs.

Note: there is no "shut down" option in the case of an ATC system as we can not rapidly bring the aircraft to a "safe state". Obviously the consequences of failure have a significant effect on the design of the system, but it is clear that many other factors are important. By far the most difficult requirements to satisfy are those of continuous operation, with little degradation in performance, such as is demanded by an ATC system.

2) FUNDAMENTAL PRINCIPLES OF FAULT TOLERANCE

2.1) Redundancy

The fundamental principle for achieving fault tolerance is to provide, within the system, redundant (spare) copies of resources which may fail, be damaged or cease to be accessible. In order to tolerate a fault we have to "replace" the damaged, faulty or inaccessible resource by one which is accessible and which is fault and error free. Examples of these resources are processors, memory and data. Data is particularly important as it is useless having spare processing power and memory in a system, if a component failure causes some of the data on which the system was working to become irretrievably lost. Much of the discussion on error recovery deals with ways of ensuring that redundant copies of data are available.

The principles of redundancy have been understood for some time, probably the earliest published work on the subject being that by von Neumann in 1956 [1]. Despite technological advances since that time many of the techniques described by von Neumann are directly applicable today.

2.2) Types of Redundancy

Redundancy may be provided either statically or dynamically. With dynamic redundancy the spare resources are not in use when the system is operating normally, but are brought into use after a component failure. With static redundancy the spare resources are in use when the system is running normally and they will be mirroring the operation of the component which they are to replace. Dynamic redundancy is usually more flexible, but the length of time taken to recover from a component failure will normally be greater than for static redundancy.

The redundant comonents may be dedicated to replacing a particular component, or be capable of replacing one of a number of components of the same type. The former type of redundancy is usually described as "one for one", or "many for one", if there is more than one spare component. With the latter type of redundancy, if we require M operational components (of one type), and have a total of N, including spares, where any of the spares can replace any of the operational components, we call this an "M out of N" structure. Usually we use M out of N structures to avoid having unnecessarily large numbers of redundant components, so we try to ensure that N << 2M.

Systems using static redundancy will obviously be of the one, or many, for one type, but dynamically redundant systems may be of either type. With a multiple processor computer we are usually trying to produce an M out of N structure, as the hardware cost of providing a spare for each processor is likely to be porhibitive. M out of N structures usually require more sophisticated, higher bandwidth, communications systems than do one for one systems, because of the need for any spare component to be able to replace any operational one. Practical systems will often incorporate a mixture of these two redundancy schemes, choosing whichever is appropriate for the component which is to be protected. Statically redundant copies of data must be made unless a cold restart (returning to the initialisation state of the data) is acceptable.

It is often important that we have time redundancy if we wish a system to be fault tolerant. Reallocation of resources in a statically redundant system may occur in a very short time, but in a dynamically redundant system the reallocation is likely to be a relatively slow process, particularly if it is performed under program control. Thus it is essential that we have time redundancy - ie that there is sufficient spare (processing) capacity in the system, for reallocation to take place, after a component failure, without causing the system to fail by not meeting some deadline. It is the lack of time for reallocation (and other aspects of recovery) that forces the "shut down" strategy to be employed with nuclear reactors. It is worth noting that time redundancy alone may be sufficient to allow some classes of faults (most notably transients) to be tolerated. The classic example of this is a communication system where packets corrupted by noise are retransmitted.

3) ANALYSIS

3.1) Hardware Reliability

The physical mechanisms underlying electronic component failure are fairly well understood and data is available on the reliability of common electronic components. The analysis of networks of components to determine the overall system reliability has been well understood for some time, and there exist a number of books on the subject ([2] for example). The models used for predicting system reliability cater for both dynamic and static redundancy, but usually with the assumption that the switching mechanisms are perfect. Often the switching mechanisms are very simple so these models yield good estimates of hardware reliability.

The analysis of the reliability of systems where failed components can be repaired and returned to the working system is also well understood, and tools for predicting this reliability exist [3]. Models which

allow the optimisation of the number of spares, given estimates of the repair time, are available. Thus it is possible to approach hardware design systematically and to make fairly good estimates of reliability, even for fairly complex systems with repair facilities.

The analysis of systems including programmable electronics, where the switching decisions may be made in software is rather less easy. However some models [4], [5], do take into account the probability that the software may not be able to recover from a hardware fault. The probability that the software can recover from a hardware fault is known as the coverage, and it is this which is the dominant factor in the reliability equations for systems which expect to withstand large numbers of component failures.

3.2) Software Reliability

Modelling software reliability is much more difficult than modelling hardware reliability, as we do not adequately understand the failure mechanisms. Software "fails" due to the presence of design faults in the programs, and a set of circumstances arising such that a faulty part of a program is executed with data which will cause it to malfunction. Thus, in order to produce an *a priori* estimate of software reliability, we must be able to produce a model of design faults, and we have to know something about the execution behaviour of the programs. The modelling of design faults - which really amounts to modelling human fallibility - seems to be particularly intractable.

There do exist a number of methods for producing *a posteriori* estimates of program reliability. These are usually based on somewhat dubious statistical estimates of the "number of bugs left in the program" given the number that have been found by testing. A summary of some of these techniques is to be found in [6], and some further references are given in [7] which is an annotated bibliography covering most aspects of software reliability.

Even if these *a posteriori* methods yielded good results, they would not help us to design systems with a predictable reliability as they can not be used until the programs have been written. The best that we can do is make some estimate of the system reliability when it is near completion. No models or tools are available to help us in the design process (to our knowledge).

It is worth noting that, because of the complexity of current LSI circuits, the problem of hardware design correctness is now becoming significant.

4) PROPERTIES OF SYSTEM ARCHITECTURES

4.1) Multi Processors

Systems which contain many processors communicating via shared main store are known as multi processors. The advantages of this type of configuration are:

1) In principle, processors can readily check each other, data sharing and data communications can readily be accomplished, and it is easy to get one processor to take over the work of another in the event of a failure;

2) There are certain applications (ie those with large databases assessed by many processes) which are naturally suited to this type of architecture. Programming these applications will be relatively easy on this type of architecture, hence the programs will be more likely to be correct, thus making the system more reliable.

The drawbacks are:

1) The inherent hardware reliability is low as a failure of the store could disable the entire system. However multi-ported store architectures, and improvements in memory chip reliability are tending to alleviate this problem;

2) The store organisation and protection mechanisms (eg capabilities) required to achieve the potential advantages are very complicated, thus reducing the inherent hardware reliability and introducing overheads which reduce the memory bandwidth. Siewiorek [8] has shown that with modern memories the control electronics often provide the dominant term in memory reliability equations, hence the introduction of complex memory control electronics is particularly disadvantageous;

3) The system throughput is limited by the memory bandwidth, and this bandwidth reduces as extra processors are added, due to contention problems, thus the maximum size of system which can usefully be configured is severely limited - exactly this problem was encountered with C.mmp at Carnegie Mellon University which has a maximum configuration of 16 processors but, for some applications, has maximum performance with only four or five processors operational. The use of store local to each processor does not entirely overcome this problem and makes the software organisation more complicated, and therefore more likely to be wrong.

4) The processors in a multi-processor system are usually physically close making them susceptible to damage.

The reliability analysis of the hardware structures of a number of extant multi processors, including C.mmp and Cm*, has been performed and the results are presented in [9].

4.2) Multi Computers

Systems where independent computers communicate via some communications medium are known as multi-computer systems. The attributes of this type of system are almost opposite to those of the multi-processor system. The drawbacks are:

1) It is difficult for processors to check one another, data communications by means of shared data items is slow and difficult, and in order for one processor to take over the job of another the relevant code must be moved or re-loaded;

2) Some applications where it is necessary to share large databases between user processes may be difficult to implement on this type of machine. If it is necessary for data sharing to be achieved by message passing between computers, then the system is likely to perform slowly. Improvements in semiconductor technology mean that the size of memory which can be configured is still increasing apace,

thus it may soon be possible to provide large enough memories on each computer that the amount of message passing is small and this objection is no longer significant (this however ignores Parkinsons law).

The advantages are:

1) The intrinsic hardware reliability is good as each computer is as independent as is possible of the others;

2) Store organisation is relatively simple and, although protection mechanisms still need to be included, these too are relatively simple.

3) The performance of the system will tend to degrade far less rapidly with the addition of extra processors than a shared store system will, but the exact performance characteristic will depend on the communications medium used, and on the application. The use of a communications medium such as Comflex [10] should ensure that the rate of performance degradation with processor addition is low.

4) Computers may be widely separated (physically) thus making this type of architecture particularly suitable for systems intended to withstand damage.

4.3) Conclusions

The attributes of these two types of architecture lead us to expect that both types of system will be used, the choice between them depending on the operational environment envisaged for the system. For systems which have to be geographically distributed it is likely that hybrid systems, that is multi computers with each computer a multi processor, will be produced. Some hybrid systems are already being developed, eg RHEA [11].

5) FAULT TOLERANCE CONCEPTS

5.1) Introduction

Terms such as failure, error and fault have been used so far on the assumption that the reader has an intuitive understanding of their meaning. It is helpful to define these terms more precisely before we consider fault tolerance in more detail. The definitions we give here stem from those used by Melliar - Smith [12], and Randell [13].

5.2) System

A system is a collection of interconnected components designed to perform a particular function. The components are items of both software and hardware, such as processes and computers. In a typical, real time, multiple processor system the set of components will include special purpose peripherals (eg radars) as well as the computing equipment. We are concerned however, with the behaviour of the computing equipment and software, not directly with the peripheral equipment (although, of course, the "peripherals" do impinge on the overall system reliabiltiy). We thus take the "Embedded Computer System" as our system for these studies. That which is not included in the system is known as the environment. Any component of the system may itself be regarded as a system under our first definition.

The system function is defined by means of a requirement which specifies the interface between the system and its environment. The requirement does not refer to the components of the system. The design specifies the implementation of the system in terms of the components, their interconnections, and the interface to the environment. The design interface specification should be equivalent to the requirement.

The requirement is assumed to be correct. This is almost undoubtedly not a valid assumption, but it is the only practical one we can make. The design is not assumed to be correct. Requirements may be drawn up for the components, but they are even less likely to be correct as they depend on design information. In practice the component requirement will probably be the best information available for judging the legality of component performance.

5.3) States

The state of a component is the (minimum) set of information necessary to characterise it at a given moment in time. The interpretation of the state depends on the component concerned, but for example, the state of a word of store would be the data value held within it. The state of a component is a function of time.

The system external state is the set of states at the interface with the environment. The system internal state is defined as the union of the external states of the system components. The internal state may obviously be defined to any level of detail required. The concept of the state of the system is useful primarily in describing the behaviour of the system. Different parts of the system do not necessarily make state transitions at the same time.

5.4) Failure

A system is said to have experienced a failure when its behaviour is such that the external state of the system, or a sequence of external states is in violation of the requirement. The failure has to be defined in terms of the requirement, (rather than the design) as we wish to include the consequences of design faults amongst the set of failures.

The failure of a component may be defined analagously. The concept is of limited theorectical use due to the limitations on the accuracy of the component requirement, but is the only practical measure of component failure which we have.

5.5) Error

We say that the system is in an erroneous state when the internal state is such that further processing by the "normal algorithms" would lead to a failure, where the failure is not entirely attributable to a subsequent fault. The erroneous state may consist of a number of errors ie states which are not legal according to the component requirement. Alternatively it may be that no particular state can be singled out as being in error, but that the collection of states in inconsistent.

The term "normal algorithms" is used to separate what are regarded as the normal algorithms from the abnormal ones, such as exception handlers. The idea of normality is subjective, and it depends on the system considered. Thus we leave the definition vague for the sake of generality.

The process of discovering that the system is in an erroneous state is known as error detection. Obviously we must be able to detect errors, and hence deduce the presence of faults, in order that a system may be fault tolerant

5.6) Fault

A fault is the mechanistic cause of an error. Examples of faults are electronic component failures (the component is being regarded as a system) and the execution of an incorrect piece of program (with data to which it is sensitive). A potential fault is a flaw in the system which will cause an error under some circumstances which are legal within the requirement of the system. All design inadequacies are potential faults.

Comprehensive classifications of faults and errors have been produced (eg by the EWICS Technical Committee on Safety and Security). It is not clear, however, that a sophisticated classification is of value. For practical purposes the only distinction we require is between those faults which we can tolerate, and those which we can not tolerate.

5.7) Error recovery

The purpose of error recovery is to restore the system to a consistent state from an erroneous one, thus avoiding the potential system failure. In order to do this it is essential to be able to identify which states are in error, or to be able to determine when the internal state is once more consistent.

There are two alternate strategies of error recovery:

Backward error recovery (BER) consists of undoing work which has already been done to return the system to a previous consistent state (Note: the state is only one which the system might legally have passed through, not necessarily one which it did pass through). After back ward error recovery it is necessary to repeat the work which has been discarded.

Forward error recovery (FER) consists of either: ignoring some (or all) of the work at hand and carrying on with only that which is believed to be correct; or correcting for erroneous behaviour by sending compensating information.

We may apply error recovery at any (component) level in a system, and usually it will be found at more than one level. We often use this facility to produce Hierarchical Error Recovery schemes in which errors which can not be corrected at one level are passed on to a superior level. This mechanism is repeated until the error is corrected or the system fails.

5.8) Fault Recovery

Fault Recovery consists of removing the faulty component from service and replacing it if necessary.

Successful error and fault recovery must be performed before a system can continue to perform its function by means of its normal algorithms.

6) HARDWARE FAULT TOLERANCE

6.1) Classical Redundancy

A Simplex (non - redundant) electronic circuit has no protection against component failures, consequently the failure of any component may lead to the failure of the whole circuit. The failure of the circuit may not be immediately apparent if there is no mechanism for monitoring the circuits behaviour.

A Duplex system (see figure 6.1) contains two identical circuits, in a static redundant configuration, whose outputs are compared. The failure of a component in either of these two circuits will lead to the system failing. The failure will be detected, as the outputs of the two circuits will disagree, but the comparison logic will not be able to determine which circuit has failed. Usually the comparison logic is much simpler than the replicated circuits so the likelihood of the comparator failing is small. It is assumed that the two circuits are independent so that common mode failures (equivalent parts of the two circuits failing at the same time) do not occur. Thus we have fairly reliable failure detection, but no capability for fault tolerance. It should be noted however that the reliability of a duplex system is less than that of the equivalent simplex system.

Adding a further redundant circuit yields the classical Triple Modular Redundant (TMR) circuit illustrated in figure 6.2. A failure in any one circuit should lead to a discrepancy between its outputs and those of the other two circuits. Thus we have achieved fault and error detection. By using majority voting, the comparator can select the correct output and send this on to the next stage in the circuit. We have thus achieved fault tolerance in perhaps its simplest form. After one failure the TMR system reduces to Duplex, so a further failure would be detected, but not tolerated. Figure 6.3 compares the reliability of the TMR system, with that of the Simplex and Duplex systems. TMR is the most reliable in the short term, but the simplex system is better once the probability of it failing during the mission exceeds 0.5.

Obviously one can continue to increase the level of redundancy provided, and certainly some quadruple redundant circuits have been produced [14]. Other enhancements to TMR have been made. For example it is quite common to replicate the voters to provide protection against voter failures. TMR stages built like this can readily be cascaded as shown in figure 6.4, giving protection against a failure of any one component in any stage. TMR with dynamic replacement of the failed component to maintain the fault recovery capability has been employed (see [15] for one example of this).

TMR is probably the most widely used fault tolerance technique, and many experimental and practical systems have been built using it, see for example [16]. The obvious disadvantages are that it is expensive as a great deal of redundancy is incorporated, and common mode (including design) faults are not detected. The use of dissimilar implementations of the components has been tried to overcome this, but this causes problems of synchronisation and error checking.

6.2) Coding and Self Checking Logic

Redundant coding of information which allows the original information to be extracted from a signal despite its partial corruption (usually by noise) has been in use for some time in communication systems [17]. This

principle has been used to protect memories in computers against changes in the stored information. A gradation in protection (like that seen with Simplex / Duplex / TMR) can be achieved. An unprotected memory will simply fail. A memory with parity checks will be able to detect a change in any odd number of bits in the memory word. A memory with an error correcting code will be able to correct some classes of error, and merely detect others. An interesting study of memory reliability is given in [8]. Codes have also been used to protect arithmetic operations [18]. At least one book on error correcting codes is available [19].

Self Checking Logic uses coded information to allow a fault in a piece of logic to be detected (but usually not corrected). The principle is that the set of combinations of coded signals generated by the circuit can be divided into two parts: legal and illegal outputs. The circuit is then designed so that an internal failure, according to some pre - defined model (usually a bit stuck at 1 or 0), will cause a legal input to be converted into an illegal output, which can therefore be detected. The detection circuit is often built on the self checking principle as well! Obviously the use of self checking logic in a Duplex system will allow it to determine which component has failed, and hence allow it to tolerate the failure. An example of a self checking logic circuit is given in [20].

6.3) Connection of Peripherals

The connection of peripherals may well be the weakest link in a fault tolerant system. Unfortunately the connection of peripherals is not always amenable to the redundancy techniques described above.

It is rarely possible to have multiple circuits all the way to the final output - eg in a control system there usually will have to be a single actuator, at the final output, on which we must rely. One counter example to this is the use of multiple actuators on aircraft control surfaces - if an actuator fails, then the remainder have sufficient power to drive the surface to its correct position.

It is also necessary to consider the interactions with peripherals if (part of) the control system fails, and then is replaced. It will be difficult to know exactly what the failed part of the system last instructed the peripheral to do, and it is possible that we may repeat an instruction in performing error recovery. We must either design the system so that we can not repeat instructions, or so that it does not matter if we do. One example of the latter strategy is to use absolute, rather than relative, position control of a servo.

Instead of having a single input, it is usually necessary to have multiple inputs, multiple homed - ie connected to different parts of the computer system, so that no single fault can cause the loss of the incoming information. This technique is used for temperature monitoring in nuclear reactors.

For error recovery it is necessary to remember what information has been sent into the system, if it is not possible for the peripheral to repeat it. Many sensors fall into this category, however some of them (eg radars) will continue to send data thus allowing the necessary information to be reconstructed, without it having been remembered by the system.

Many computer peripherals store information redundantly, and are capable of detecting a significant proportion of their own failures. This is especially true of the magnetic media, thus it is quite possible to produce

a Duplex filestore which is reasonably fault tolerant. The use of magnetic media in a statically redundant system can be difficult due to their non - deterministic response time. The outputs from Duplex discs will not stay exactly in step, and they may even perform seeks in different orders, hence they must either be artificially synchronised, or the voter must be capable of performing the checking correctly despite the asynchrony. The Tandem System chooses to synchronise the discs.

Despite efforts made to protect the system against peripheral failures, the peripherals will still often be the limiting factor in the reliability which can be achieved.

6.4) Other Techniques

There are three other techniques of hardware redundancy which seem to be particularly promising.

The first technique is that of using a "hardcore" - a small part of the system which is highly fault tolerant, and hopefully design correct, which is responsible for the fault tolerant operation of the rest of the system. By concentrating on the design of the hardcore, and simply making the rest of the system as modular, and mutually independent as possible, it should be possible to achieve a very high coverage. This technique has been used in the JPL STAR [15], where the hardcore is TMR "Test and Repair" processors, with dynamic replacement of failed processors.

The second technique exploits the ability of microprogram, in a microprogrammed processor, to monitor very closely the behaviour of the processor, and hence to detect faults. This technique is known as micordiagnostics [21]. It seems natural, in building a fault tolerant processor, to make the "microengine" the hardcore, and then to use micro-diagnostics to monitor the rest of the processor.

The final technique endeavours to exploit the power of LSI techniques to achieve protective redundancy "on chip". The complexity of LSI circuits is such that we can no longer assume that they are design correct, so it is desirable to provide alternate circuit designs. Integrated circuits are rather prone to common mode failures - transistors made by the same process tend to fail under similar conditions. Withe current I.C. packing densities, the pin count, rather than the chip area is normally the limiting factor. Consequently the provision of distinct designs of the same circuit on chip seems to be an attractive way of solving these two problems. Experimental circuits with on chip redundancy have been produced, but they have not been used in commercial machines [22]. These circuits use complementary logic (ie all signals are inverted, ANDs replace ORs etc.) for the redundant design, thus ensuring detection of certain classes of errors.

6.5) Conclusions

Many and varied techniques are known for the design and implementation of fault tolerant hardware. It is now possible to build hardware which is extremely reliable. Therefore probably the most important aspect of hard-ware design is the provision of adequate support to the protection and error recovery mechanisms required by the software.

7) RECOVERY

7.1) Fundamental Prerequisites

The more rapidly an error is detected, the easier it is to determine which component is faulty, and hence to perform fault recovery. The less errors are allowed to spread (the better they are contained), the easier it is to perform error recovery, as fewer errors have to be corrected. Thus rapid error detection and good error containment are fundamental to successful recovery.

Error detection in hardware and software are complementary, as each will tend to detect different classes of error. However the less error detection there is in the hard / firm ware, the more the software will have to perform with consequent reduction in system performance, and probably worse error detection. If one is forced to use conventional hardware, then the proportion of error checking performed by the software will have to be very high.

7.2) Software and Hardware Faults

Most systems built to date have been designed to recover from hardware faults, but there is now increasing interest in producing systems which can survive software faults. Software faults have to be treated by the use of forward error recovery, or the provision of redundant programs (or part of programs) of different designs. This does not significantly effect the error recovery algorithms, and often the main difficulty in recovery will be in deciding where the fault occurred, and hence whether a hardware or software unit should be replaced.

7.3) Basic Functions of Recovery

There are four primary functions which the recovery system has to perform:

Identify the faulty Component;
"Replace" the faulty component;
Identify the errors;
"Correct" the errors.

They are not necessarily performed either explicitly, or in that order.

If the error detection and containment mechanisms are good, then the faulty component may be unambiguously identified as a by - product of the error detection mechanisms. This is obviously the case with TMR system. If the faulty component is not identified by the error detection mechanism, then action must be taken to find it. This can either be done explicitly, eg by running checks on all the suspected components (routining), or implicitly as a side effect of the error recovery.

Replacement of the faulty component will require resource reallocation in a dynamic redundant system. If the software is at fault the reallocation may only be temporary (see section 8.3). In a static redundant system it will probably not be necessary to do anything - the reallocation is effectively performed as part of the error detection mechanisms.

Again if the error detection and containment mechanisms are good, then they will have identified the errors. If this is not the case then we can try to check that the states of components, which we think may have been affected by the fault, are error free. With a static redundant system this is conceptually easy - we simply check against the spare components - but it may be quite time consuming in practice. For a dynamic redundant system we have to try to establish consistency by other means, if this is feasible (imagine running a total consistency check on a large, disc based, data base). Very often consistency checks will not be feasible, and the set of errors will only be found by subsequent error detection.

Error correction in the static redundant case is probably a null action. In the dynamic redundant case the erroneous state will have to be corrected. This is usually done by taking copies of the state of the system components from time to time whilst it is running normally, and reverting to the latest of these copies after an error has been detected. This technique is known as checkpointing. Obviously this form of error connection will cause some work to be lost. Checkpointing is fairly straightforward in a mono - processor, mono - programming system, but it becomes more difficult when parallel processes have to be considered. Some of the problems of multiple process checkpointing and error recovery are considered in section 9. In a dynamic redundant system error recovery will often have to be of a heuristic nature (trial and error) since it is not in general possible to know when all the errors have been corrected. It should be noted that other error correction strategies are possible, in particular forward error correcting techniques may be used.

It should be clear that error recovery in static redundant systems is much easier than in dynamic redundant systems. However considerable effort is now being put into recovery techniques for dynamic redundant systems, because of the economic desirability of this type of system. In choosing whether to have a statically or dynamically redundant system, one is trading hardware cost against the complexity of, and time taken to perform, error recovery.

7.4) Types of Error Recovery

Two contrasting techniques of error recovery, are those of "backward" and "forward" recovery as defined in section 5. The prime difference between these two techniques is that, for FER to be performed, knowledge of the system function is required, whereas BER can be performed without such knowledge. Because of the need for knowledge of the system function, the mechanisms for performing FER can not be provided by the operating system, but those for BER can be. Using BER it is conceptually possible to provide a "recoverable virtual machine" to the users of the system, which would hide the recovery operations from them. Thus the users could produce applications programs without being aware that the recovery mechanisms are there, which is obviously an attractive way of achieving reliability. Some work has been carried out towards this end [23].

One other important distinction between BER and FER, is their behaviour in the presence of design faults. FER will give some automatic protection against design faults, for example, if a transaction fails due to the presence of a program flaw, descarding the transaction and going on to the next one (without correcting the program) will usually suffice to overcome the fault. With BER the transaction would be repeated, and would fail in exactly the same manner, unless the faulty program was changed or replaced. Obviously BER in conjunction with a scheme for

replacing the faulty piece of program could enable design faults to be tolerated (see section 8).

Another important aspect of recovery mechanisms is whether they operate with centralised, or decentralised, control. The use of centralised control implies that we are confident about the correct working of the controller for (at least) the duration of the recovery action. A centralised recovery mechanism would obviously be appropriate in a system with a hardcore. With decentralised control the components of the system work with some degree of mutual suspicion, and often the decision making process will be replicated. This type of recovery might be used in an M out of N system, where it is not always certain that the faulty component has been found, and hence it is inadvisable to trust one particular component. Here recovery decisions could be made independently by three different components, the results sent to all the affected components, and a decision only implemented if all the versions of it agree.

A further influence in the choice between centralised and decentralised recovery is the fault model for the system. If the model is single fault, single error, then we can safely use centralised control as we know that the decision making component is trustworthy for the duration of the recovery action (assuming our fault model to be valid). Conversely, if the behaviour of the system is such that further faults can occur, or errors be detected, during recovery then a decentralised approach is most likely to be satisfactory.

A third classification of recovery types is relevant for systems in which resource reallocation may occur. The reallocation may be determined statically - ie a predetermined reconfiguration for predefined failure modes, or dynamically - ie the recovery mechanisms decide wthat resource reallocation is appropriate as and when the faults occur. The static method has the benefit of simplicity but with the obvious drawback of inflexibility. In a complex system the number of alternative configurations which must be stored could be very large, which is a further disadvantage. In some programming systems the mapping of the programs onto the hardware is specified in the program source text, and here, static reallocation is the only possible solution (unless we are willing to edit and recompile all the software during error recovery!).

Dynamic resource reallocation allows more flexible recovery, but is probably more difficult to achieve than static recovery. A dynamic approach must be taken where the system load (ie the set of programs which are running) varies with time, and can not be predicted in advance. Dynamic resource reallocation is only possible if the application software is organised so that parts of the program can be moved without having to be changed. This requirement can be viewed as the multiple processor extension to the meaning of relocatable code. This form of relocatability can either be provided implicitly, as in the Cm* Algol68 system [24], or explicitly as proposed in [25].

7.5) Examples of Recovery from Hardware Faults

A detailed case study of recovery from hardware faults is given in the paper on dual matched processors, however it is worth mentioning some other systems here, and making some general comments.

The JPL STAR [15] mentioned previously is an example of a system using a hard core. It uses a very low level of software technology and requires the application programmer to make explicit use of the machine

facilities to store the state of his programs from time to time. A description of the software recovery mechanisms is given in [26].

Pluribus is a more recent system currently being used as an ARPANET node, and it is an example of an M out of N system. This also uses low level programming techniques, in which strips of code are produced to run for a predetermined time. Error detection is performed by checking the running time of the strips. A description of the system is to be found in [27].

The PP250 [28] is a more sophisticated system which makes heavy use of capabilities. It is programmed in a high level language (Coral 250) oriented towards the use of the capability structure. This system is one of the few fault tolerant computers available commercially.

8) RECOVERY FROM SOFTWARE FAULTS

8.1) Introduction

A software fault is the event of executing an incorrect piece of program with data which exposes this incorrectness (we assume that the program works correctly under most circumstances, and that the failure is not a consequence of hardware malfunctions). It is a matter of taste whether we say that a program which fails because it was given data which is outside its specification is itself at fault (for not checking the data before operating on it), that the generating program was at fault, or that the specification is at fault for allowing these inconsistencies. Regardless of how we apportion the blame it is clear that recovery from a software fault requires either getting rid of the data which caused the fault and carrying on with the other work required of the system, or processing it with a program which is correct (for that data). The former method is clearly a type of FER so we will not attempt to discuss it further (as it is application dependent). The latter method can use either static redundancy, or dynamic redundancy and BER. We will describe one example of each of these two types of recovery.

8.2) N Version Programming

N version programming is an example of static redundancy. A number of different versions of a program are written and they all run (in parallel) and their outputs are compared before any action is taken. This is obviously the software equivalent of TMR. If one of the versions of the program disagrees with the others it will (probably) be unnecessary to take it out of service, as it is likely that it will later agree once more with the other versions. The only error recovery which is likely to be necessary is to make the data used by the faulty program version consistent with that used by the other versions, perhaps by copying the data used by one of the other versions. Two experiments with N version programming are described in [29] and [30].

There are a number of serious problems associated with this technique. The processing power required for running this type of program is about N times that for the simplex program. Similarly the amount of effort required to write the programs is N times higer. The redundancy does not protect against common mode failures, eg failures due to incorrect specification. There will often be considerable difficulty in designing the voter as the different versions of the program may produce answers which are "sufficiently close" to be acceptable, but which are not

identical. In some cases it may be difficult to design different versions of the program (especially if we have additional constraints such as speed of execution). There will often be difficulties in synchronising the different versions of the program especially if there are interactions with asynchronous events. N version programming does not help in resolving problems of interactions between parallel processes (see section 9).

8.3) Recovery Blocks

Recovery Blocks [31] are an example of dynamic redundant programming. They allow alternative strategies to be provided which are tried in turn until one is successful or until all the alternatives are exhausted. They have been implemented as an extension to the block structure in sequential PASCAL. The typographical form is:

```
ENSURE
    acceptance test
BY
    normal algorithm
ELSEBY
    first alternative
ELSEBY
    second alternative

        .
        .
        .
ELSEERROR;
```

In operation the normal algorithm is executed, and copies are made of any data which the algorithm changes. When the normal algorithm terminates the acceptance test is evaluated, and if the test is satisfied the block is exited. If the test fails then the copies of the data made whilst the normal algorithm was executing are used to return the data to the state it was before the algorithm was executed. The first alternative is then tried, and so on. If none of the alternatives are successful then the block terminates with an error exit. Recovery blocks are intended to be used hierarchically, as are the blocks in a conventional programming language, so the error exit passes as an error to the immediately superior recovery block. This is a good example of hierarchical error recovery.

It is interesting to consider the ways in which recovery blocks have been implemented. The initial implementation [32] was entirely software based, and made copies of the changed data on a heap. The additional software to copy the data, and to manage the information on the heap, was generated by the modified PASCAL compiler, thus the user of the system was unaware of the work which was being done on his behalf. The main drawback of this system is that it is very slow, due to the need to monitor all the operations which update data, hence a second implementation employing hardware assistance has been produced.

The hardware assistance consists of a "recovery cache" [33] which largely replaces the code which monitors the updating operations and the heap data storage. The recovery cache works by intercepting memory accesses on the processor / memory bus and copying data which is going to be changed into its own internal memory. To restore the memory state after an acceptance test has failed, the cache uses the addresses and data which it has stored to restore the values which were current before the faulty algorithm executed. The recovery cache virtually eliminates the performance degradation seen with

the purely software implemented version of recovery blocks. This is one of very few examples of hardware being designed to support the needs of the error recovery software.

Many of the drawbacks to the recovery block scheme are the same as those for N version programming. The problems of synchronisation, and use of large amounts of processor power do not arise, but a lot of additional store and programming time are required. Acceptance tests are not always easy to produce, and may possibly be of similar complexity to the algorithms which they are trying to check, and hence are as prone to error. Again no help is given with interacting parallel processes, but this problem is being studied.

8.4) Program Correctness

There is a further problem of software redundancy schemes which is inherent to the concept. If one produces N versions of a program (or part thereof) it will require more effort than producing one version (probably N times more). It is arguable, but very difficult to prove, that one could produce a more reliable end product by devoting all the extra effort to trying to get the original program right. It is not at all clear how to resolve this argument, or indeed that there is a general solution to it.

One possible way around the problem is to show formally that the programs which have been produced are correct. This is currently far beyond our ability in program proving, and program specification, for any realistic set of programs and is likely to remain so for some time. In the interim, despite the enormous cost and inadequacies, it is likely that redundant programming techniques will be used particularly in small, critical, systems such as reactor safety systems.

9) MULTI PROCESS RECOVERY

9.1) Introduction

Recovery in computer systems, using dynamic redundancy and BER, where there are parallel processes is difficult because the interactions between the processes affect the process states, and hence recovery of one process may require recovery of another. Figure 9.1 shows the classical example of this problem known as the "Domino Effect". Recovery of either process by returning to an earlier state requires the state of the whole system to be returned to the initialisation state.

The techniques for producing fault tolerant software which we described earlier do not help with this problem although they are quite applicable to faults which are local to a component. Practical solutions to genuine multi process recovery do not exist (to the authors knowledge) - it appears that in systems with parallel processes it is assumed that the processes can be recovered independently, or some measure of FER is used to overcome interaction problems. This is a good strategy to use, if it is practicable, as it will greatly simplify the recovery process, but, unfortunately, this is not usually possible.

Theoretical models of multi - process recovery, which could in principle be implemented, have been developed. The following sections describe a model based on Occurrence Graphs (loosely the graph which one gets when executing a Petri net) and some of the practical problems of implementation.

9.2) Occurrence Graphs and Recovery

The use of occurrence graphs (O.G.s) to model error recovery in a distributed system was first described by Merlin and Randell [34]. In their paper Merlin and Randell describe an error recovery protocol, known as the "Chase Protocol", and prove that it will return the system to a consistent state after an error or errors, under certain (not very restrictive) assumptions. This protocol can be used without stopping any parts of the system which are not affected by errors so long as the error messages propagate faster than those generated by the normal execution of the system. Rather than reproduce the proofs here, we merely describe the salient features of O.G.s, and the Chase Protocols.

O.G.s comprise three elements: places, represented by circles; bars, represented by vertical lines; and arcs, with the obvious representation. Places represent the state of components, bars represent events in the system which cause state changes, and the arcs show mechanistic causality. If it is possible to restore the state of a particular component to one of its previous states, then the corresponding place is known as a "recoverable place", and is represented by a pair of concentric circles. An example of an O.G. (for some unspecified system) is shown in figure 9.2.

To recover from an error (in graph terms) requires a set of recoverable places to be established which represents a consistent set of states for the components of the system (as defined by the arcs in the graph). This set of places is known as a recovery line. If the system has stopped executing, the recovery line can be found fairly easily in a centralised manner. The Chase Protocols define a mechanism whereby a recovery line can be established under decentralised control, whilst the system is still executing.

In essence the protocols require messages to be sent along the arcs of the graph, from a place or bar which is thought to be incorrect. The messages sent forwards (in the forward direction of the arcs) are passed on by the place or bar which receives them, then the place or bar is effectively destroyed. The messages are passed forward until all the graph elements which they can reach have been destroyed. The messages passed backwards cause bars and non-recoverable places to be destroyed, but are not propagated by recoverable places. Thus the messages stop at the recovery line. Merlin and Randell show that the minimum amount of work is lost. After the Chase Protocols have been executed, the system can be restarted from the set of states defined by the recovery line.

9.3) Practical Recovery

Implementation of a recovery scheme based on O.G.s and the Chase Protocols requires the provision of a mechanism for generating the O.G., and for making safe copies of the component states. This operation is known as checkpointing (although other terms such as producing an "adult trail" are commonly used). The graph should be stored so that a fault which may affect a particular component, will not cause the corresponding part of the graph to be destroyed. The most obvious way of doing this is to build the graph on a semi - permanent medium, such as a disc, and multiple copies (static redundancy) can be employed to guard against failure of the storage medium. The details of how the information is stored, and how much is stored (ie what is the minimum amount of information necessary to reconstruct a component state) depend heavily on the application.

There are certain problems associated with O.G.s which render the direct implementation of the Chase Protocols difficult.

1) Unless the graph is built systematically, recovery may involve an arbitrarily large amount of work being lost (as the nearest recovery line may be arbitrarily "far away").

2) The Chase Protocols tacitly assume that the graph has been built correctly, but this structure is as prone to error as any other in the system (although, of course, one is very careful to try to generate it correctly and then protect it).

3) The graph will continue to grow indefinitely, unless something is done to remove elements when they are no longer needed. The difficulty is to know then they are of no further use (since one can always synthesize some suitably devious error, which requires the place which has just been deleted in order that recovery may take place).

It is possible to limit very strictly the amount of work that is lost, be ensuring that every interaction between the components (at the level of abstraction which we are considering) causes the generation of a recoverable place for the components involved. This a very costly exercise, and probably gives finer recovery than is required. It would be desirable to generate fewer recoverable places, giving coarser recovery, but with less overheads during normal running. In order to do this we have to recognise larger scale "atomic" actions, so that we build a graph which contains recovery lines at fairly well bounded "intervals". This problem is being studied [35].

Some attempt has been made to address the problem of graph correctness and deletion [36]. The Chase Protocols have been extended to cater for some classes of corruption to the graph, and a systematic way of deleting the graph elements has been evolved. This latter mechanism does not completely solve the problem of deciding when information is no longer needed, but does reduce the probability of discarding useful information.

10) CONCLUSIONS

10.1) Area of discourse

The paper presents a very broad, hence somewhat shallow, view of reliable computer system design. It has tried to provide a fairly pragmatic guide to redundancy and recovery, with very little excursion into reliability theory, or into a philosophical treatment of the concepts of fault, error etc. This approach was chosen partly because material is available on these two aspects of reliability (for example [2], and [13] respectively), and partly because of the authors predilections.

Obviously many aspects of reliability have been missed out altogether, and no attempt has been made to say how one should go about designing a reliable system. However it is hoped that the material has been presented in a sufficiently general way to make the principles clear, and so that the prospective designer of a reliable system will at least know what some of his problems are likely to be, and what solutions are available.

A great deal is known about the design of reliable systems, particularly statically redundant systems, but there are obviously a large number of unsolved problems. We certainly can not fulfil our initial aim of showing that a design will meet its requirement for reliability, and it seems unlikely that this will be possible in the near future. The final section

of this paper indicates which problems, in the authors view, are the most worthy of consideration.

10.2) Research Problems

Techniques for achieving tolerance to software faults are crude and inefficient, and, due to the nature of the problem, they are very likely to remain so. Thus techniques which help in the production of correct (fault intolerant) programs are important. All the work on programming language design, compiler design techniques, program specification and program verification techniques is relevant.

Hardware design techniques will also benefit from a better understanding of specification and verification techniques. When the mechanisms required by the software for checkpointing, error detection, error containment, exception handling and error recovery are better understood, it should be possible to provide assistance to these mechanisms by appropriate hardware design.

A better understanding of the concept error should enable better error detection mechanisms to be developed, with many consequent benefits. One hopes that it will become possible to provide a form of redundancy in the state of complex components (such as processes) which will allow error detection to be performed as readily as is possible for a binary word encoded with parity.

The ability to better specify the consistency required between the states of different components should also allow improved error detection. In many systems it is not necessary that all the data is completely consistent all the time. If it were possible to formalise the "degree of consistency" required, then it would help us in determining when components could be considered independent, and hence their interactions ignored for the purpose of error recovery. The ability to decide how to partition data to increase this independence would be very valuable.

The design of reliable systems is normally approached in an ad hoc manner. The provision of formal methods of specifying the reliability requirements, and analysing the design for compliance to these requirements, would greatly help in systematising the design process. Obviously the provision of design tools would be particularly advantageous.

The most important contribution which could be made to reliability theory would be the production of an *a priori* model of design faults. It is very unclear how to do this, even if it is possible.

11) References

[1] "Probalistic logics and the synthesis of reliable organisms from unreliable components", J von Neumann, in Automata Studies, C E Shannon and J McCarthy eds., Princeton University Press, 1956.

[2] "An elementary guide to reliability", Dummer and Winton, Pergamon.

[3] "ARIES - An automated reliability estimation system for redundant digital structures", Y Ng, A Avizienis, Proc. Reliability and Maintainability Symposium, 1977.

[4] "Reliability modelling techniques for self-repairing computer systems", W G Bouricius, W C Carter, P R Schnieder, Proc. of the 24th National Conference of the ACM, 1969.

[5] "A reliability model for gracefully degrading and repairable fault tolerant systems", Y Ng, A Avizienis, Digest of papers from FTCS7, 1977.

[6] "Applicability of statistical software reliability models for reactor safety software verification", S Bologna, W Ehrenberger, Report RT/ING(79)1, Comitato Nazionale Energia Nucleare, 1979.

[7] Reliable Software: A selective annotated bibliography, T Anderson and S K Shrivastava, Software Practice and Experience, Vol. 8, 1978.

[8] "The effect of semiconductor chip failure modes on system reliability and performance", D P Siewiorek and S Elkind, Proc. FTCS 8, 1978.

[9] Reliability Modelling of Multiprocessor Architectures, R Joobbani, D P Siewiorek, Proc. 1st Int. Conf. on Distributed Computers, Huntsville Alabama, 1979.

[10] "COMFLEX - a high speed packet switch for inter - computer communication", J A McDermid, Proc. of Eurocomp 78, 1978.

[11] "RHEA: a damage and fault tolerant digital communication support system for distributed avionic processing", D R Powell, Proc. 1st Int. Conf. on Distributed Computers, Huntsville Alabama, 1979.

[12] "Software Reliability: the role of programmed exception handling", M P Melliar-Smith, B Randell, Proc. ACM Conf. on Language design for reliable software, 1977.

[13] "Reliable Computer Systems", B Randell, P C Lee, P A Treleaven, in Lecture Notes in Computer Science No. 60: Operating Systems - An Advanced Course, G Goos, J Hartmanis Eds., Springer Verlag, 1978.

[14] "Primary processor and data storage equipment for the Orbital Astronomical Observatory", T B Lewis, IEEE Trans. on Electronic Computers, EC-12, 1963.

[15] "The STAR (Self Testing and Repairing) Computer: An investigation of the theory and practice of fault tolerant computer design", A Avizienis, G C Gilley, F P Mathur, D A Rennels, J A Rohr, D K Rubin, IEEE Trans. Comp. vol. C-20, 1971.

[16] "Microcomputer Reliability Improvement using Triple Modual Redundancy", J F Wakerley, Proc. IEEE vol. 64, No. 6, 1976.

[17] "A class of codes for signalling on a noisy continuous channel", J L Kelly, IRE Trans., IT-6, 1960.

[18] "Arithmetic Error Codes: Cost and effectiveness studies for application in digital system design", A Avizienis, IEEE Trans, Comp. vol. C-20, 1971

[19] "Error Correcting Codes", W W Peterson, E J Weldon, MIT Press, 1972.

[20] "Design of Self - Checking Checkers for Berger Codes", M A Marouf, A D Friedman, Digest of papers from FTCS8, 1978.

[21] "System Modelling and Testing Procedures for Microdiagnostics", C V Ramamoorthy, L C Chang, IEEE Trans. Comp. vol C-21, 1972.

[22] "Fault Tolerance of a general purpose computer implemented by Very Large Scale Integration", R M Sedmak, H L Liebergrot, Digest of papers from FTCS8, 1978.

[23] "Redundancy and Recovery in the HIVE virtual machine", J M Taylor, Proc. European Conf. on Soft. Syst. Eng. 1976.

[24] "Programming Issues raised by a multiprocessor", A K Jones, R J Chansler, I Durham, P H Felier, D A Scelza, K Schwans, S R Vegdahl, Proc. IEEE Vol. 66 No. 2, 1978.

[25] "POSER - A Process Organisation to Simplify Error Recovery", J A McDermid, RSRE Memo. 3249, 1979.

[26] "STAREX Self-Repair routines: Software recovery in the JPL-STAR computer", J A Rohr, Digest of papers from FTCS3, 1973.

[27] "Pluribus - A reliable multiprocessor", S M Ornstein, W R Crowther, M F Kraley, R D Bressler, A Michel, F E Heart, Proc. of National Computer Conference 1975, 1975.

[28] "Reliability Assurance for System 250, a reliable real time control system", C S Repton, Int Conf on Computer Communications 1972.

[29] "N version programming: a fault tolerance approach to reliability of software operation", L Chen, A Avizienis, Digest of papers from FTCS8, 1978.

[30] "Software diversity in reactor protection systems: An experiment", L Gmeiner, U Voges, Proc. of Safecomp 79, 1979.

[31] "Recovery Blocks in action: a system supporting high reliability", T Anderson; R Kerr, Proc. Int. Conf. on Software Engineering, 1976.

[32] "Concurrent pascal with backward error recovery: Implementation", S K Shrivastava, Software - Practice and Experience Vol. 9 No. 12, 1979.

[33] "A recovery cache for the PDP11, P A Lee, N Ghani, K Heron, Report No. TR134, Computer Laboratory, University of Newcastle upon Tune, 1979.

[34] "Consistent state restoration in distributed systems", P M Merlin, B Randell, Report No. TR113, Computer Laboratory University of Newcastle upon Tyne, 1977.

[35] "A formal model of Atomicity in asynchronous systems", E Best, B Randell, Report No. TR130, Computer Laboratory, University of Newcastle upon Tyne, 1978.

[36] "Checkpointing and Error Recovery in distributed systems", J A McDermid, RSRE Memo, 3271.

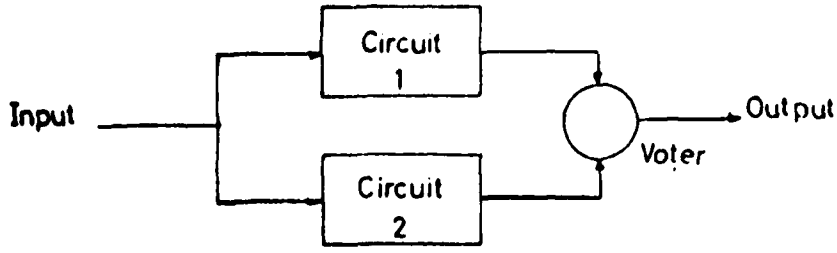
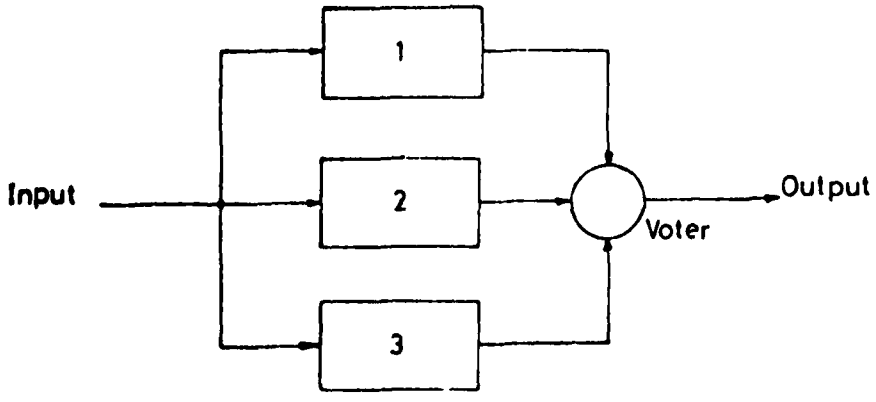
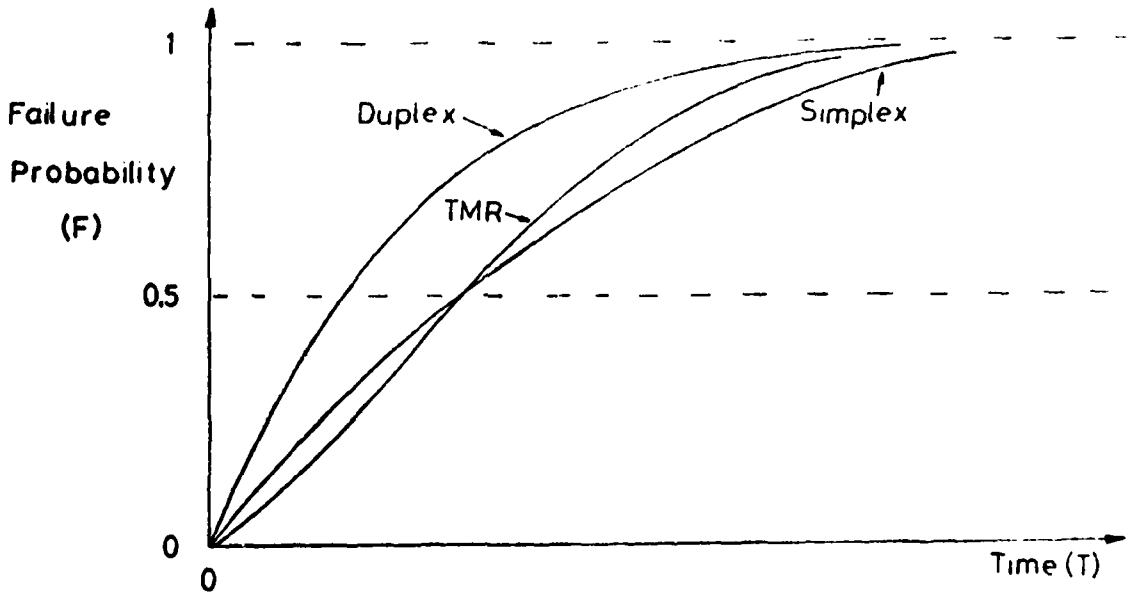


Fig. 6.1 Duplex Circuit



Redundant Circuits

Fig. 6.2 TMR



Simplex: $F = 1 - e^{-\lambda T}$

Duplex: $F = 1 - e^{-2\lambda T}$

TMR: $F = 1 + 2e^{-3\lambda T} - 3e^{-2\lambda T}$

where λ is the failure rate of the simplex circuit

Fig. 6.3 Comparison of redundant structures

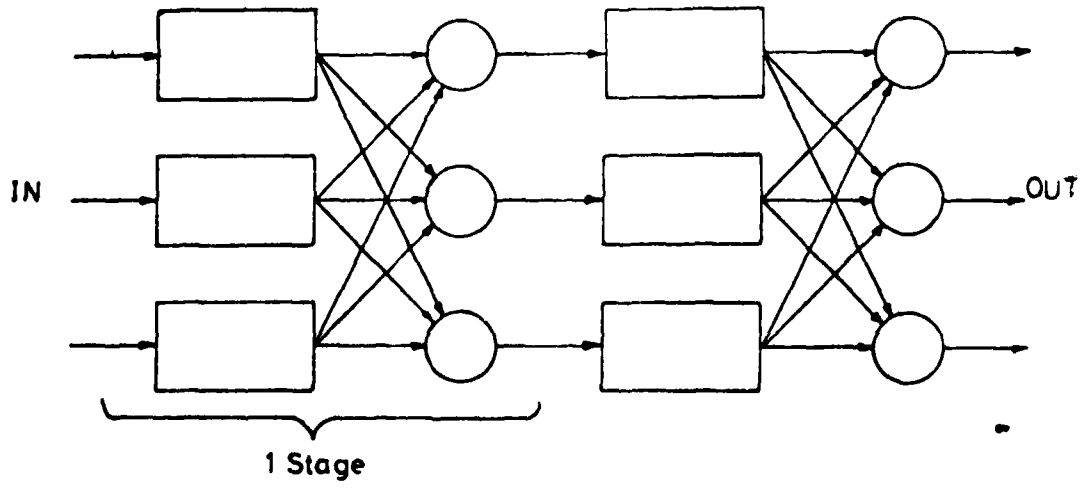


Fig. 6.4 Cascaded TMR Stages

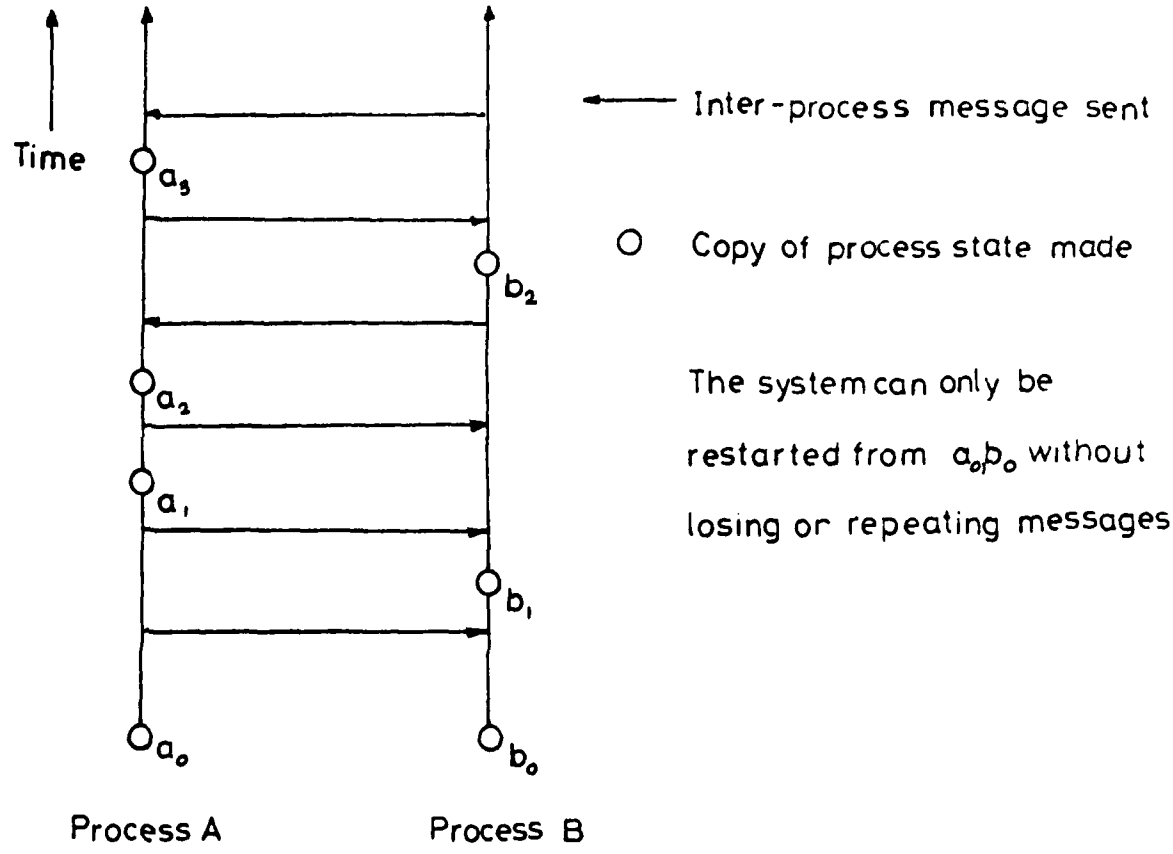


Fig. 9.1 The Domino Effect.

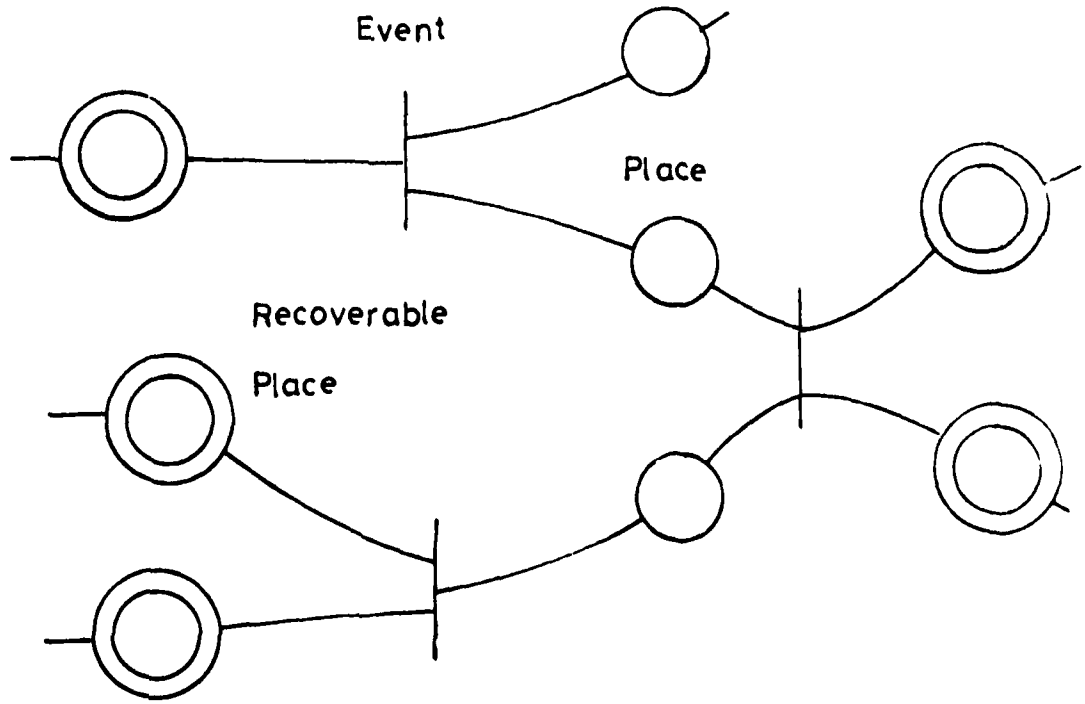


Fig. 9.2 An Occurrence Graph

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Report 80003	3. Agency Reference	4. Report Security Classification
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location RSRE		UNLIMITED
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

An Overview of Reliable Computer System Design

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, Initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
McDermid, J A				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This paper was produced to support a series of lectures on reliable computer system design ~~given at a NATO ASI summerschool~~ on multiple processor computers. The paper was intended to be fairly self contained, but it does lack a description of a practical fault tolerant system. This omission was made because one of the other lectures (the one on the dual matched processor system referred to in section 7.5) analysed a typical system in some detail. Perhaps the best example of a practical fault tolerant system is the "Tandem 16" (see overleaf) multi computer produced by Tandem Computers Limited. Any other sins of omission or commission are unitentional.

continued on separate piece of paper

Abstract continued

The paper presents an overview of reliable computer system design. It attempts to provide a pragmatic guide to redundancy and recovery, but does not give a very thorough discussion of either the theory or philosophy of reliable systems. The paper introduces and defines the basic concepts of reliability, and describes the basic mechanisms for achieving fault tolerance. It compares the attributes of multi processor and multi computer systems from the point of view of reliability. It describes in some detail techniques for achieving tolerance to both hardware and software faults. The paper concludes by outlining some of the major unsolved problems of reliable system design.

The hardware is described in:

"A Fault - Tolerant Computing System", by J A Katzman,
and the software in:
"A Non - Stop Operating System", by J F Bartlett.
Both these papers were presented at the Hawaii International Conference on System Sciences in 1978.